

# Real-time Operating Systems and Systems Programming

Compilation and Utilities  
Virtual Memory

# Compilation steps

- Source code
- Preprocessing
- Compiler
  - Assembly code
- Assembler
  - Object code
- Linker

# Why is Awareness Needed?

- Error message source discovery
- Assembly code checking
- Makefile creation

# GCC options

- Preprocess only: -E
- Compile only: -S (gives assembly code)
- Skip linking: -c

Example

# Makefiles

- Compilation must be an atomic process
  - Otherwise the programmer debugs larger chunks
- Save time on compiling large projects
- Help with modularity
- Compile unfamiliar programs without thinking

# Makefile layout

- File uses tabs instead of spaces
- Named either "makefile" or "Makefile"

```
target: prerequisite1 prerequisite2  
    commmand
```

```
myprog: myprog.c myprog.h  
    gcc myprog.c -o myprog
```

# Laying out a program

- Modules:
  - Spread the program over several .c files
  - Use .h files for function prototypes and data
- For .h:

```
#define _header_h_
#ifdef _header_h_
...
#endif
```

# .h files

- Describe the "interface"
- Function prototypes
- Data types and structures declared
- `const` and `#define`
- `#includes` for other headers



# Makefile with separate linking

- Simple makefile which compiles in several steps
- Note first and last directives

```
# Makefile for the sample
sample: sample.o my_math.o
        gcc -o sample sample.o my_math.o
sample.o: sample.c my_math.h
        gcc -c sample.c
my_math.o: my_math.c my_math.h
        gcc -c my_math.c
clean:
        rm sample *.o core
```

# Makefile (2)

- Make checks upon running the command whether it needs to compile anything by looking at file dates and their dependencies
- So it tries to only compile the minimal set

# clean Convention

- Makefiles often specify (and programmers expect) a way to clean out the temporary files  
make clean  
clears the files if specified
- If for some reason you need to recompile and make does not want to:  
touch filename.h

# Implicit rules

- Make can compile when some rules are omitted
- It "knows" how to compile from .c to .o, for example, if the names match and only target and prerequisites are present

# Implicit Rule Example

```
objects = main.o kbd.o command.o display.o \  
          insert.o search.o files.o utils.o
```

```
edit : $(objects)  
      cc -o edit $(objects)
```

```
main.o : defs.h  
kbd.o : defs.h command.h  
command.o : defs.h command.h  
display.o : defs.h buffer.h  
insert.o : defs.h buffer.h  
search.o : defs.h buffer.h  
files.o : defs.h buffer.h command.h  
utils.o : defs.h
```

```
# Note special .PHONY keyword here!!!  
.PHONY : clean  
clean :  
      rm edit $(objects)
```

# .PHONY

- Make clean does not have prerequisites and thus will always run
- If someone makes a file named "clean" into directory, cleaning will fail
- .PHONY tells that we are dealing with a command, not a target file

# Macros

- You can define macros in a makefile to avoid repeating yourself
- Macros are defined as:  
    name = value
- Used as:  
    \$(name) or  
    \${name}

# Multiple directories

- Sometimes you need to split program modules into directories
- Every module has its own makefile
- Program has a directory for every module and one for all of the .h files
- Main Makefile creates the program
- Makefiles in modules make the corresponding object files



# Directory Example

- C program uses Stack module and Queue module and has a main.
- Program has 7 files: StackTypes.h, StackInterface.h, QueueTypes.h, QueueInterface.h, StackImplementation.c, QueueImplementation.c and Main.c
- The target is a program in a directory which contains subdirectories Stack, Queue and Include (containing every .h file)

# Stack dir

- StackImplementation.c and the makefile:

```
export: StackImplementation.o

StackImplementation.o: StackImplementation.c \
                        ../Include/StackTypes.h \
                        ../Include/StackInterface.h
        gcc -I../Include -c StackImplementation.c
# substitute a print command of your choice for lpr below
print:
        lpr StackImplementation.c
clean:
        rm -f *.o
```

# Queue dir

- QueueImplementation.c and the makefile:

```
export: QueueImplementation.o
    QueueImplementation.o: QueueImplementation.c \
        ../Include/QueueTypes.h \
        ../Include/QueueInterface.h
    gcc -I../Include -c QueueImplementation.c
# substitute a print command of your choice for lpr
    below
print:
    lpr QueueImplementation.c
clean:
    rm -f *.o
```

# Notes

- -I (capital i) tells where the library includes can be found; use commas for multiple; don't use spaces
- This enables us to gather .h files in one location for ease of reference
- The \ symbol before line-end escapes it.

# Main directory

- Main includes main.c and makefile:

```
export: Main
Main: Main.o StackDir QueueDir
    gcc -o Main Main.o ../Stack/StackImplementation.o \
        ../Queue/QueueImplementation.o
Main.o: Main.c ../Include/*.h
    gcc -I../Include -c Main.c
StackDir:
    (cd ../Stack; make export)
QueueDir:
    (cd ../Queue; make export)

#continues
```

# Main directory (2)

```
print:
    lpr Main.c
printall:
    lpr Main.c
    (cd ../Stack; make print)
    (cd ../Queue; make print)

clean:
    rm -f *.o  Main  core
cleanall:
    rm -f *.o  Main  core
    (cd ../Stack; make clean)
    (cd ../Queue; make clean)
```

# Notes

- Unix command sequence in brackets makes them run as a subprocess
- So the directory changes apply, but only for the subprocess itself

# Let's Add Macros

```
CC = gcc
HDIR = ../Include
INCPATH = -I$(HDIR)
DEF = $(HDIR)/StackTypes.h $(HDIR)/StackInterface.h
SOURCE = StackImplementation
export: $(SOURCE).o

$(SOURCE).o: $(SOURCE).c $(DEF)
            $(CC) $(INCPATH) -c $(SOURCE).c
print:
            lpr $(SOURCE).c
clean:
```



# GNU Make

- GNU Make has a ton of features such as:
  - Control structures and conditional clauses, cycles
  - Simple text modifying features
  - Automatic variables referring to target/source
- Gmake manual:  
<http://www.gnu.org/software/make/manual/make.html>

# GNU autotools

- <http://www.gnu.org/autobook/>
- Makefile does not work well with portable applications for different Unixes
- Thus automake and autoconf are used
- Programmer writes Makefile.am and configure.in files
- Those are changed to configure and Makefile.in
- Configure makes Makefile using the latter

# Makefile.am

- Describes program and its requirements on a general level

```
## Makefile.am -- Process this file with automake to produce  
Makefile.in  
bin_PROGRAMS = foonly  
foonly_SOURCES = foo.c foo.h nly.c scanner.l parser.y  
foonly_LDADD = @LEXLIB@
```

# configure.in

- Like this:

```
dn1 Process this file with autoconf to produce a configure script.
```

```
AC_PREREQ(2.59)
```

```
AC_INIT([foonly], [2.0], [gary@gnu.org])
```

```
AM_INIT_AUTOMAKE([1.9 foreign])
```

```
AC_PROG_CC
```

```
AM_PROG_LEX
```

```
AC_PROG_YACC
```

```
AC_CONFIG_FILES([Makefile])
```

```
AC_OUTPUT
```

# Usage of autotools

- Usually:

```
aclocal  
autoconf  
automake  
./configure  
make  
make install
```

- Distribution:

```
make dist
```

creates xxx.tar.gz with readied configuration

# Don't forget

- gdb
  - and (somewhat) graphical ddd
- hexdump
- objdump

# Page fault

- ♦ As there is more virtual memory than real memory, we must swap pages between "real" and "backup" memory
- ♦ It's called paging
- ♦ Page fault: an attempt to read memory which is not in the RAM
- ♦ When page fault happens, the couple of milliseconds needed for memory access suddenly take a far greater amount of time
- ♦ Hard disks become noisy

# How to get memory

- ♦ There are two ways of getting memory
  - upon starting your program (exec) when your program gets its memory space and is allocated space in there for its constants, code text and stack space
  - in your program:
    - auto variables
    - malloc
    - mmap: map a file into virtual memory
  - fork: *copy on write*
- ♦ When program stops, its memory space collapses



# Tracing memory

- ♦ You can trace memory allocation using

```
void mtrace(void);  
void muntrace(void);
```

- ♦ Use environment variable named MALLOC\_TRACE to specify the file which will store the statistics about memory allocation and release
- ♦ The first activates, the second deactivates trace
- ♦ GNU specific: mcheck.h provides it
- ♦ Result is not human-readable – use a command:

```
mtrace programname mtrace-log
```

# Valgrind

- Memory debugging and profiling tool
- Makes your program really slow, but documents it while it runs
- Usage:  
    `valgrind --tool=memcheck prog args`
- Tools: memcheck, callgrind, cachegrind
- For callgrind run `callgrind_annotate`

# mmap()

- ♦ mmap() maps a file into virtual memory (or creates an anonymous mapping)
- ♦ Sometimes useful:
  - We can read only parts of file which we use
  - mmap() lets you write changes back to disk
  - we can open files larger than mem+swap

```
void * mmap (void *address, size_t length, int protect,  
             int flags, int filedes, off_t offset)
```

- Parameters: desired start of mapping, length, protection data, management data, file descriptor and file offset

# mmap() parameters

- ♦ prot: PROT\_READ, PROT\_WRITE, PROT\_EXEC bits
  - depending on system: *write* is usually *read* or write protected files can not be written when PROT\_READ is missing
- ♦ flags: refine mapping:
  - MAP\_PRIVATE: don't write changes into file
  - MAP\_SHARED: changes visible in file and other processes
  - MAP\_FIXED: get this address or fail
  - MAP\_ANONYMOUS: don't open a file (some systems expand heap using this trick)

# mmap.c Example

# `munmap()` & `msync()` & `madvise()`

- `munmap()`: removes mapped space starting from an address to given address (may remove several); can handle unmapped segments.
- `msync()`: write mapping to file from given point
- `madvise()`: suggests how you want to use an address region: for random access, sequential access; will we need it all eventually or is the contents becoming irrelevant and when anything happens to it, the client won't leave the room in screaming agony.